Market-based Autonomous Application and Resource Management in the Cloud

Stefania Costache, Samuel Kortas, Christine Morin, Nikos Parlavantzas

Market-based Autonomous Application and Resource Management in the Cloud

Stefania Costache , Samuel Kortas , Christine Morin, Nikos Parlavantzas

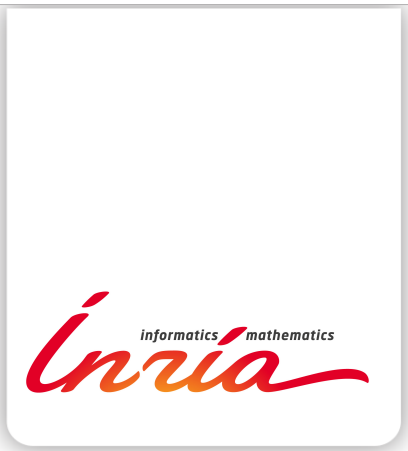
RESEARCH REPORT

N° 8648

December 2014

Project-Team MYRIADS

ISSN 0249-6399 ISRN INRIA/RR--8648--FR+ENG



Market-based Autonomous Application and Resource Management in the Cloud

Stefania Costache *, Samuel Kortas *, Christine Morin†, Nikos Parlavantzas ‡ §

Project-Team MYRIADS

Research Report n° 8648 — December 2014 — 36 pages

Abstract: Managing private High Performance Computing (HPC) clouds, although it has different advantages due to improved infrastructure utilization and application performance, remains difficult. This difficulty comes from providing concurrent resource access to selfish users who might have applications with different resource requirements and Service Level Objectives (SLOs). To overcome this challenge, we propose Merkat, a market-based SLO-driven cloud platform. Merkat relies on a market-based model specifically designed for on-demand fine-grain resource allocation to maximize resource utilization and it uses a combination of currency distribution and dynamic resource pricing to ensure proper resource distribution. To scale the application's resource demand according to the user's SLO, Merkat uses autonomous controllers, which apply adaptation policies that: (i) dynamically tune the amount of CPU and memory provisioned for the virtual machines in contention periods or (ii) dynamically change the number of virtual machines. Our evaluation with simulation and on the Grid'5000 testbed shows that Merkat provides flexible support for different application types and SLOs and good user satisfaction compared to existing centralized systems, while the infrastructure resource utilization is improved.

Key-words: cloud computing, resource management, autonomous systems

§ The authors are in alphabetic order.
* EDF R&D, Clamart, France
† INRIA Rennes-Bretagne Atlantique, Rennes, France
‡ INSA, Rennes, France

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Gestion autonome des ressources et des applications dans un nuage informatique selon une approche fondée sur un marché

Résumé : Les organisations qui possèdent des infrastructures de calcul à haute performance (HPC) font souvent face à certaines difficultés dans la gestion de leurs ressources. En particulier, ces difficultés peuvent provenir du fait que des applications de différents types doivent pouvoir accéder concurremment aux ressources tandis que les utilisateurs peuvent avoir des objectifs de performance (SLOs) variés. Pour résoudre ces difficultés, cet article propose un cadre générique et extensible pour la gestion autonome des applications et l'allocation dynamique des ressources. L'allocation des ressources et l'exécution des applications sont régies par une économie de marché respectant au mieux des objectifs de niveau de service (SLO) tout en tirant parti de la flexibilité d'un nuage informatiqueé et en maximisant l'utilisation des ressources. Le marché fixe dynamiquement un prix aux ressources, ce qui, combiné avec une politique de distribution de monnaie entre les utilisateurs, en garantit une utilisation équitable. Simultanément, des contrôleurs autonomes mettent en œuvre des politiques d'adaptation pour faire évoluer la demande en ressource de leur application en accord avec les objectifs (SLO) fixés par l'utilisateur. Les politiques d'adaptation peuvent : (i) adapter dynamiquement leur demande en terme de CPU et de mémoire pour les machines virtuelles en période de contention pour l'obtention de ressources (ii) et changer dynamiquement le nombre de machines virtuelles. Nous avons évalué cette plate-forme par simulation et sur l'infrastructure Grid'5000. Nos résultats ont montré que cette solution: (i) offre un support flexible aux applications de différents types ayant des demandes variés en terme de niveau de service; (ii) augmente l'utilisation des ressources de l'infrastructure; (iii) conduit à une meilleure satisfaction des utilisateurs par rapport aux solutions centralisées existantes.

Mots-clés : nuages informatiques, allocation de ressources, systèmes autonomes

1 Introduction

HPC infrastructures, composed of a large number of computers, are acquired and used by an increasing number of organizations. The efficient use of these infrastructures is a key challenge for many of these organizations, which strive to minimize the cost of maintaining the infrastructure while satisfying the constraints of applications running on it. Recently, to ease their infrastructure's management, some of these organizations choose to virtualize and transform them in "private clouds", managed by specialized frameworks [20, 36, 43, 30, 18]. The main difficulty that these resource management frameworks need to address is allocating the resources needed for each application.

First, resource allocation has to be done with respect to the infrastructure's capacity limitations. Having enough capacity to meet all user requests in the highest demand periods is rarely the case, as expanding the resource pool is expensive. To differentiate between user requests in these periods, most of these frameworks [20, 36, 43, 18] rely on priority classes, i.e., users are given a priority class for their applications. However, in this case the users might abuse their rights and run less urgent applications with a high priority, taking resources from users who really need them.

Second, resource allocation has to be done by considering the application and user requirements. Some applications might have a resource demand that varies frequently, based on the load they need to process, e.g., web applications, while others can simply adapt their demand to the current resource availability, e.g., bags of tasks or MapReduce applications. At the same time, while some applications can achieve better performance by scalling horizontally, i.e., scaling the number of nodes, others might benefit from scaling their resource demand vertically, i.e., scaling the resource amount per node [4]. Disregarding these characteristics can lead to poor resource utilization and application performance. Finally, users might have different SLOs for their applications. Some users want the application results by a specific deadline, (e.g., a user needs to send her manager the output of a simulation by 7am the next day) or as soon as possible (e.g., a developer wants to test a newly developed algorithm). Using the cloud computing's "on-demand" provisioning model of virtual resources to manage the organization's infrastructure is an attractive approach to deal efficiently with the variety of application resource models and user requirements. By provisioning virtual machines dynamically, advanced Platform-as-a-Service (PaaS) solutions [3, 6, 12, 32] can satisfy application performance requirements, but, *while only some of them provide vertical and horizontal scalling, none of them properly address the case of contention periods.*

In this paper we present the design and evaluation of *a platform for application and resource management in private clouds*, called Merkat. Merkat applies a unique approach to multiplex the limited infrastructure capacity between applications with different resource demand models while maximizing the infrastructure utilization and providing support to meet different SLOs. In Merkat each application runs in an autonomous environment composed of a set of virtual machines (VMs) customized to its needs. This environment, which we call a virtual platform, *elastically adapts its resource demand to meet the user's SLO.* The resource demand of the virtual platforms is regulated through the use of a proportional-share market that provides fine-grained resource allocation and a dynamic pricing model which varies with the total infrastructure

demand. The market's allocation scheme distributes resources in terms of CPU and memory in a fine-grained manner to the provisioned VMs, *allowing applications to scale their resource demand not only horizontally but also vertically.* The market's currency distribution policy and dynamic price *ensure proper resource utilization in contention periods*, by favoring users who get the most value from the resources. On top of the implemented market, the virtual platforms use two policies to adapt the application's resource demand to the current resource availability and price: (i) a vertical scaling policy that adapts the application resource demand per VM; (ii) a horizontal scaling policy that adapts the application resource demand in terms of number of VMs. These policies can also be combined for better application execution. Through the use of the market, each virtual platform takes resource demand adaptation decisions independently from the others. This resource control decentralization makes Merkat *flexible in supporting multiple application models and SLOs.*

We evaluated Merkat in simulation and on a real testbed. We implemented the proportional-share market in CloudSim [8]. We evaluated the performance of the proportional-share market in terms of total user satisfaction when applications adapt their resource demands per VM to track a user-given SLO [14]. We also tested Merkat on the Grid'5000 [7] testbed with two application types: static MPI applications and malleable task processing frameworks [13]. Our results show that: (i) Merkat is flexible, allowing the co-habitation of different applications and policies on the infrastructure; (ii) Merkat increases the infrastructure resource utilization, through vertical and horizontal scaling of applications; (ii) Merkat has low performance degradation compared to a centralized system that supports a fixed SLO type; this degradation is caused by the decentralized nature and the application selfish behavior.

The remaining of this paper is organized as follows. Section 2 describes our approach. It introduces the principles behind it and the resource management process. Section 3 describes the vertical and horizontal policies implemented in Merkat. Section 4 presents evaluation results and Section 5 discusses limitations and future directions of improvement. Section 6 describes the related work. Finally, Section 7 concludes the paper.

2 Merkat

Merkat has been designed to manage the clusters owned by an organization, which needs to run HPC workloads but it cannot use public cloud resources, due to security constraints, e.g., the data that needs to be processed is too sensitive. This is often the case of organizations carrying out research activities in computational sciences. This case is supported by Electricite de France (EDF), which relies on HPC simulations to optimize the day-to-day electricity production or choose the safest and most effective configurations for nuclear refuelling. The clusters are shared among a multitude of users (e.g., scientific researchers) which might come from different departments (e.g., production, development) and might need to use specific frameworks or libraries to run their applications. These users not only have different SLOs for their applications, e.g., getting computation results until a specific deadline, or executing the application as fast as possible, but they might also want to assign different importance degrees to their requests.

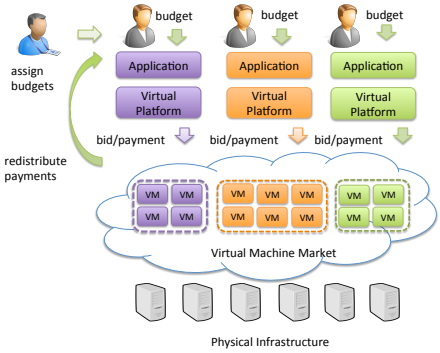


Figure 1: Overview of the Merkat system.

Merkat meets the user demands, while also allowing the organization to get the best out of its infrastructure, by maximizing its resource utilization. Figure 1 gives an overview of Merkat. Merkat relies on: (i) a virtual machine market to allocate resources and internal currency to manage the priority of users' demands and (ii) a set of virtual platforms that manage and elastically scale applications to meet user SLOs.

To be able to run applications on the infrastructure, users are assigned budgets by an administrator in the form of a virtual currency. Merkat disposes of a total amount of currency, which is distributed among uses based on defined weights. We call the currency unit a *credit*. A user will receive a budget of credits proportional to its weight in the system. It is the task of the administrator to add/remove users to/from the system, set up and adjust the total amount of currency and the users' weights. Virtual currency is desirable when managing a private infrastructure; because no external currency is introduced, price inflation is bounded.

When a user wants to run an application, she assigns it a replenishable amount of credits, called application budget, and sends a request to Merkat to start a virtual plaform for it. This application budget reflects the maximum cost the user is willing to support, or the true priority, for running an application. These budgets are replenished automatically at a system-wide interval defined by the administrator. The replenishment is defined as transfering a user-defined amount from her account to the application's budget. Merkat ensures that the amount with which the budget is replenished never leads to a total budget that exceeds the initial budget amount. Replenishable budgets are used to minimize the risk of depleting the application's budget in the middle of the application's execution.

A virtual platform runs the user's application by acquiring VMs from the virtual machine market. The design of the virtual platform is specific to the application type the user wants to run. VMs are acquired by submitting payments for their resources, also called bids. Bids can be scaled up or down during the VM runtime. This mechanism has several advantages. First, it provides *flexibility for designing a variety of policies* to adjust the resource allocation of the application in a selfish way, with regard only to the user's SLO and application budget. This selfish behavior is natural, as users care only about the perfor-

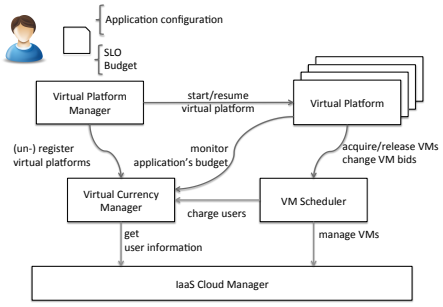


Figure 2: Prototype's components and their interactions.

mance of their applications. In this context, currency management allows some control over this selfish behavior and gives users incentives to use better the infrastructure, while also allowing flexibility in meeting their SLOs. Second, allowing a bid per resource per VM leads to *efficient resource utilization* for the organization, as there is a strong incentive for users to design policies that avoid oversubscription by requesting just as much as the application uses from the capacity of a VM. Lastly, communicating resource demands through the use of bids leads to *a generic system*, capable of supporting any kind of application and SLO.

2.1 Architecture

Merkat is composed of three main services, the VM Scheduler, the Virtual Currency Manager and the Virtual Platform Manager. Figure 2 gives an overview of them. The *Virtual Currency Manager* applies virtual currency distribution policies and manages user and application budgets. The *VM Scheduler* is in charge of allocating resources to running VMs and computing their node placement. The used algorithms are described in Section 2.2. The *Virtual Platform Manager* acts as an entry point for users to run their applications on our system. To start an application on the infrastructure, a user submits a request containing a virtual platform template to the Virtual Platform Manager. The virtual platform template contains information regarding the application controller, adaptation policy parameters, including desired user SLO, and the configuration of the VMs, e.g., VM disk image. We define the SLO as the performance objective the user wants for her application, e.g., a specific execution time or throughput. To start the virtual platform, the Virtual Platform Manager checks any initial deployment conditions that a user has specified. For example, the user might want to start running an application only if the resource price is below a threshold. If these conditions are not met, the deployment of the virtual platform is either postponed or canceled. For example, the deployment is canceled if the application needs to start its execution before a given deadline (the user's defined SLO) and the price is too high to allow it. If these conditions are met, the Virtual Platform Manager creates the virtual platform and registers it with the Virtual Currency Manager. The Virtual Platform Manager is also in charge of resuming any virtual platforms that might suspend their VMs to avoid executing applications in high price periods.

To manage the users and the VMs, Merkat interacts with an IaaS Cloud Manager [39]. The IaaS Cloud Manager provides interfaces to start, delete and migrate VMs, manage their storage, network, and moreover, to keep information about the infrastructure's users.

2.2 The Virtual Machine Market

To allocate resources to VMs based on the value of the submitted bids, Merkat uses a proportional-share policy. Originally, this policy was used by the operating system schedulers to allocate CPU time to tasks proportionally to a given weight, and inversely proportionally to the sum of all the other concurrent task weights [45]. Merkat uses a modified version of this policy, in which each provisioned VM has an associated bid for its resources, i.e., CPU and memory, and it receives an amount of resource proportional to the bid and inversely proportional to the sum of other concurrent bids [24]. The value of the bid of a VM can be changed during the VM runtime.

This policy is advantageous as it allows provisioning VMs with arbitrary resource allocations at a small algorithmic complexity (O(N)), aspect which becomes important with the increasing scale of the infrastructures. Moreover, this policy is easy to understand and use as it avoids starvation and unvoluntary VM shutdown or preemption. Because each VM receives a resource amount, even in high price periods, policies can be designed to allow applications to decide whether to adapt to these small allocations or voluntary shutdown some of their components.

In Merkat, the implementation of the proportional-share policy follows four steps: (i) VM bid submission; (ii) VM allocation computation; (iii) VM placement; (iv) price computation.

2.2.1 VM Bid Submission

To provision VMs, a bid, in the form of a vector $b =< b_{cpu}, b_{memory} >$ needs to be submitted for their resources, CPU and memory, to a central entity, called VM Scheduler. This bid can be submitted directly by the user or the virtual platform that manages the VMs. The initial bid can be computed based on past price history and the user's current budget. In our implementation we compute the VM bid based on the current resource price. The bid submitted for a VM is persistent: the user can specify it when the VM is started and the VM Scheduler will consider this value in its allocation decisions during the VM runtime. Nevertheless, the value of the bid can be further changed to cope with price fluctuations.

2.2.2 VM Resource Allocation

The VM Scheduler periodically computes resource allocations for the VMs for which a bid was submitted, by considering the value of their bid and a resource utilization cap, a_{max}, i.e., the maximum resource utilization of a VM during its lifetime. The VM Scheduler uses the a_{max} values to distribute free resources to other VMs needing them. We consider that the user can estimate a_{max} for her VMs.

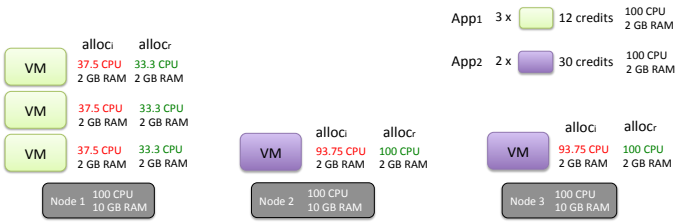


Figure 3: An example of resource fragmentation. 2 applications requesting 3 and respectively 2 VMs run on 3 nodes. If resources are allocated with the proportional-share policy from the entire infrastructure capacity, the CPU allocations for the first application's VMs cannot be guaranteed. If resources are allocated with the proportional-share policy from the node capacity, allocations can be guaranteed and all available resources are distributed.

The resource allocation computation is performed in two steps: (i) to ensure maximum utilization, the VM Scheduler computes the VM allocation considering the entire infrastructure as a single physical node; (ii) to cope with the resource fragmentation, i.e., the infrastructure capacity is divided among nodes, the VM Scheduler corrects the allocations to fit to the node capacity.

When considering the entire infrastructure as a single physical node, the VM Scheduler computes the resource allocations as follows. For each resource, given a set of n resource bids $b_i(t)$, with $i \in \{1..n\}$, for a time interval t and a capacity of C units, the resource allocation for each bid $b_i(t)$ is equal to:

$$a_i(t) = \frac{b_i(t)}{\sum_i^n b_i(t)} \cdot C, \tag{1}$$

However, as the infrastructure capacity is partitioned between nodes, there are situations when resulted allocations cannot be enforced. This issue is illustrated in Figure 3. We consider 3 nodes with a capacity of 100 CPU units and 10 GB of RAM each. We also consider 5 VMs with a maximum resource utilization of 100 CPU units and 2 GB RAM each. The first 3 VMs receive a bid of 12 credits, and the last 2 VMs receive a bid of 30 credits. In this example, only the CPU resource is a bottleneck and each VM will receive the maximum amount of requested RAM. Using Equation 1, the VM Scheduler computes an allocation $alloc^j_{iCPU}$ (depicted with red in Figure 3), $j \in 1..5$, as follows: 37.5 CPU units for the first 3 VMs and 93.75 CPU units for the last two VMs, where 1 CPU unit represents 1% of CPU time. Practically, these allocations cannot be enforced.

To solve this issue, the VM Scheduler corrects the allocations by recomputing them after placing the VMs on the nodes and using the capacity of the node in Equation 1. In the previous example, the resulted allocation $alloc^j_{rCPU}$ (depicted with green in Figure 3), $j \in 1..5$, is: 33.3 CPU units for the first 3 VMs and 100 CPU units for the last 2 VMs. The resulted allocation difference is called *allocation error* and is defined as follows:

$$e_r = \frac{\left|alloc^i - alloc^r\right|}{alloc^i}, \tag{2}$$

As seen in the next section, this allocation error is used in placing the VMs on nodes.

2.2.3 VM Load Balancing

When VM requests are received the VM Scheduler places them initially on the nodes with the lowest resource utilization. To minimize the VM allocation error, the VM Scheduler might migrate VMs between nodes. The process of migrating VMs among nodes is called load balancing. As having a high number of migrations leads to a performance degradation for the applications running in the VMs, the load balancing process tries to make a trade-off between the number of performed migrations and the VM allocation error. For example, it won't make sense to migrate a VM when its allocation error is 1%. To select the VMs to be migrated at each scheduling period, the VM scheduler relies on an algorithm based on a tabu-search heuristic [17]. Tabu-search is a local-search method for finding the good solutions of a problem by starting from a potential solution and applying incremental changes to it.

Algorithm 1 details the load balancing process. The algorithm receives the list of current nodes, *nodes*, the list of running VMs, *vms*, the list of VMs to be started at the current scheduling period, *newvms*, and three thresholds: (i) maximum number of iterations performed by the algorithm to obtain a better placement than the current one, N_{iter}; (ii) maximum allocation error supported for the VMs in their current placement, E_{max}; (iii) maximum number of migrations required to reach a better placement, M_{max}. Based on this information the algorithm computes the new placement of VMs on nodes and outputs a migration plan, composed of the VMs to be migrated, and a deployment plan, composed of the VMs to be started.

The algorithm starts from the current VM placement and tries to minimize the VM allocation error while keeping the number of VM migrations within the given limit M_{max}. If new VMs need to be created, they are placed on the least loaded nodes (Lines 6-8) before the VM placement is improved. Then, the VM placement is incrementally improved by placing the VM with the highest allocation error among the CPU and memory resources to the node that minimizes it (Lines 15-20). Note that, as each VM has two allocated resources, the maximum allocation error is the maximum among the computed error for each resource (Line 15). The VM allocation error computation is performed by a method called $ComputeErrors$. To avoid being stuck in a suboptimal solution, the algorithm uses a list that memorizes the last changes (Line 21).

The N_{iter} threshold is used to ensure the algorithm finishes: if there is no improvement in the last N_{iter} iterations, the algorithm stops.

2.2.4 Price Computation

In our approach we compute the resource price as the sum of all bids divided by the total infrastructure capacity. If this price is smaller than a predefined price, i.e., reserve price, then the reserve price is used. Users are charged based on their allocated resource amounts.

Algorithm 1 VM load balancing algorithm.

ComputePlacement $(nodes, vms, newvms, N_{iter}, E_{max}, M_{max})$
$migrationPlan \leftarrow \emptyset$ // *migrations to be performed*
$deploymentPlan \leftarrow \emptyset$ // *deployments to be performed*
$nIterations \leftarrow 0$ // *number of iterations until an improvement*

for $vm \in newvms$ **do**
  $node \leftarrow$ least loaded node from $nodes$
  $node.vms \leftarrow node.vms \cup \{vm\}$
$solution_{old} \leftarrow nodes$ // *current placement of VMs*
$solution_{best} \leftarrow nodes$ // *new placement of VMs*
$tabu\_list \leftarrow \emptyset$ // *list of forbidden moves*
$e_{worse} \leftarrow$ inf
e = ComputeErrors(vms, nodes)
while $nIterations < N_{iter}$ and $e_{worse} > E_{max}$ **do**
  $(vm, e_{max}) \leftarrow$ vm with $e_{max} = \max_{1 \leq i \leq n} \max\{e_{ic}, e_{im}\}$, $vm \notin tabu\_list$
  $source \leftarrow vm.node$
  $destination \leftarrow$ node which minimizes e_{max}, $node \notin tabu\_list$
  $vm.node \leftarrow destination$
  $source.vms \leftarrow source.vms - \{vm\}$
  $destination.vms \leftarrow destination.vms \cup \{vm\}$
  $tabu\_list \leftarrow tabu\_list \cup \{(vm, source)\}$
  e = ComputeErrors(vms, nodes)
  $e'_{max} \leftarrow \max_{1 \leq i \leq n} \max\{e_{ic}, e_{im}\}$
  $nMigrations =$ count number of migrations required to reach the new placement
  if $e_{worse} - e'_{max} > 0$ and $nMigrations < M_{max}$ **then**
    $solution_{best} = nodes$ // *Keep the best solution so far*
    $e_{worse} \leftarrow e'_{max}$
    $nIterations \leftarrow 0$
  else
    $nIterations \leftarrow nIterations + 1$
for $node \in solution_{old}$ **do**
  for $vm \in node.vms$ **do**
    if $vm.node \neq node$ and $vm \notin newvms$ **then**
      $migrationPlan \leftarrow migrationPlan \cup (vm, vm.node)$
for $vm \in newvms$ **do**
  $deploymentPlan \leftarrow deploymentPlan \cup (vm, vm.node)$

return $(migrationPlan, deploymentPlan)$

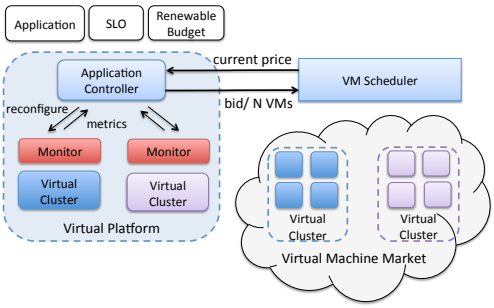


Figure 4: Virtual Platform overview.

2.3 Virtual Platforms

When running the application on the Merkat's virtual machine market, it is not sufficient for the virtual platform logic just to acquire a number of VMs and compute their bids at the beginning of the application execution. As the resource price and allocation are dynamic, this logic will not guarante that the application receives the right amount of resources to efficiently meet its SLO. If the price increases, the current user budget might not be enough to cover the cost of the needed resources. Thus, the application will run with less resources and not only will the SLO not be met, but also the budget will be wasted. If the price decreases, more resources could be provisioned, or the user could spend less for the already acquired resources. These issues are solved by the virtual platform, which monitors the application performance and adapts autonomously to the resource price and user requirements.

2.3.1 Virtual Platform Architecture

Figure 4 illustrates the architecture of a virtual platform. A virtual platform is composed of one or more virtual clusters, and an application controller, that manages them on behalf of the application. The application controller receives as input a virtual platform template, containing the application description and adaptation policies. An application can have one or more components, requiring different software configurations. Thus, for each application component, the application controller deploys a virtual cluster and starts the application component in it. We define a virtual cluster as a group of VMs that have the same resource configuration and use the same base VM image. As the application's components might have different performance metrics, a different virtual cluster monitor running user specific monitoring policies can be started in each virtual cluster.

During the application runtime, the application controller checks the application's performance metrics and adapts dynamically the virtual platform resource demand to the infrastructure resource prices and reconfigures the application. The user can interact with her application controller during its execution to modify her SLO, the budget for the application execution, or to retrieve statistics regarding the application.

3 Application Adaptation Policies and Use Cases

We demonstrate the case of virtual platforms in Merkat by proposing and applying two policies to adapt the resource demand of an application: vertical and horizontal scaling. These policies use the dynamic resource price as a feedback signal regarding the resource contention and respond by adapting the application resource demand given the user's SLOs.

Both policies address three cases:

- *Preserve budget when the SLO is met:* When the SLO can be met, the virtual platform reduces its resource demand and thus its execution cost. The remaining budget can be used afterwards to run other applications.
- *Provide more resources when the SLO is not met:* When the application requires more resources to meet its SLO, and its budget affords it, the virtual platform increases its resource demand.
- *React when the SLO is not met due to budget limitations:* A last case that is considered is when the application cannot meet its SLO, because the current resource price is too high and the application budget is too limited to ensure the desired resource allocation. In this case, we consider that the user has two options: (i) stop the execution of her application; (ii) or continue to run it with the same amount of resources.

The policies are run periodically and use two performance thresholds, upper and lower, as a trigger: when the application performance metric crosses the thresholds, the policy takes an action that changes the virtual platform resource demand.

3.1 A Vertical Scaling Policy

The vertical scaling policy controls the amount of CPU and memory resources allocated to a VM as well the amount that is payed for them. This policy computes periodically the resource bids for each VM based on the following information: (i) current, minimum and maximum VM resource allocation; (ii) current application performance metrics, v; (iii) application reference performance, v_{ref}, upper and lower performance thresholds, v_{high} and v_{low}; (iii) the current resource bids; (iv) the value of the last bid change, $last\_bid\_change$; (v) and the budget to be spent for the next time period, bid_{max}. Algorithm 2 describes this policy.

The policy works as follows. *To preserve the user budget when the SLO can be met*, the policy decreases the resource bids in two situations: (i) if the performance metric drops below the lower threshold (e.g., the remaining execution time of the application is less than 75% of remaining time to deadline); (ii) or if the allocation per VM for one resource reaches the maximum (Line 5). *To provide more resources when the SLO is not met*, i.e., when the performance metric is above the upper performance threshold, the policy increases the resource bids (Line 13). *If the current budget is not enough to meet the SLO*, the policy decides between two options, based on the application and SLO type: (i) it suspends the application; (ii) it recomputes the bids to favor the resource with a small allocation. The first option is taken for batch applications, when

Algorithm 2 Vertical scaling policy

1: **VerticalAdaptation** (bid, bid_{min}, $last\_bid\_change$, bid_{max}, $alloc$, $alloc_{min}$, $alloc_{max}$, v, v_{ref}, v_{low}, v_{high})
2: $resources \leftarrow \{cpu, memory\}$
3: $T \leftarrow \left|\frac{v_{ref}-v}{v}\right|$
4: **for** $r \in resources$ **do**
5: **if** $(v < v_{low})$ and $alloc[r] > alloc_{min}[r]$ **then**
6: $bid_{tmp}[r] \leftarrow max(bid[r]/max(2, 1+T), bid_{min})$
7: **if** $last\_bid\_change < 0$ **then**
8: $\delta \leftarrow |bid[r] - bid_{tmp}[r]|$
9: **if** $\left|\frac{\delta - |last\_bid\_change|}{\delta}\right| < 0.1$ **then**
10: $\delta \leftarrow \delta/2$
11: $bid_{tmp}[r] \leftarrow max(bid_{tmp} - \delta, bid_{min})$
12: $bid[r] \leftarrow bid_{tmp}[r]$
13: **if** $(v > v_{high}$ and $alloc[r] < alloc_{max}[r])$ or $(alloc[r] < alloc_{min}[r])$ **then**
14: $bid_{tmp}[r] \leftarrow bid_{tmp}[r] \cdot max(2, (1+T))$
15: **if** $last\_bid\_change > 0$ **then**
16: $\delta \leftarrow |bid[r] - bid_{tmp}[r]|$
17: **if** $\left|\frac{\delta - |last\_bid\_change|}{\delta}\right| < 0.1$ **then**
18: $\delta \leftarrow \delta/2$
19: $bid_{tmp}[r] \leftarrow bid[r] + \delta$
20: $bid[r] \leftarrow bid_{tmp}[r]$
// bids are re-adjusted due to budget limitations
21: **if** $bid[memory] + bid[cpu] > bid_{max}$ **then**
22: $w \leftarrow \emptyset$
23: **if** $alloc[r] \geq alloc_{max}[r]$, $r \in resources$ **then**
24: $w[r] \leftarrow 0$
25: $bid_{max} \leftarrow bid_{max} - bid[r]$
26: **else**
27: $w[r] \leftarrow 1 - \frac{alloc[r]}{alloc_{max}[r]}$, $r \in resources$
28: **for** $r \in$ resources with w[r]$\neq 0$ **do**
29: $bid[r] \leftarrow \frac{bid_{max}}{\Sigma w[r]} \cdot w[r]$
30: return bid

the current price is too high to continue their execution, i.e., the VM allocation becomes less than $alloc_{min}$. In this case, the policy will resume the application when the price drops, e.g., VMs can receive 75% of their maximum allocation at the current budget. The second option is taken if the application cannot be suspended. In this case, the application continues running with less resources than needed to meet the SLO. Thus, the policy recomputes the bids as a part from the application budget that is proportional to the difference between the actual and maximum allocation of the VM (Lines 21-29).

The value with which the bid changes is given by the "gap" between the current performance value and the performance reference value: $T = |(v_{ref} - v)/v|$. A large gap allows the application to reach its reference performance fast. To avoid too many bid oscillations the policy uses the value of the past bid change in its bid computation process. For example, if the bid was previously increased and at the current time period the bid needs to be decreased with a similar value, the bid oscillates indefinitely. Thus, the algorithm decreases the current

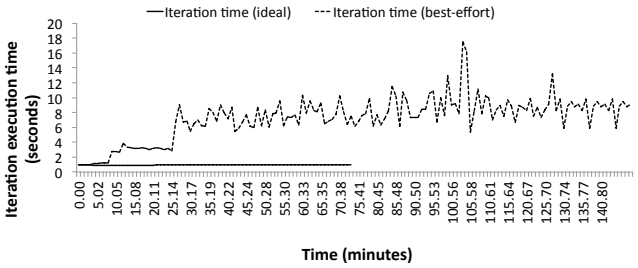


Figure 5: Application execution time variation with a best-effort controller.

bid with half of its value (Lines 8-10 and 16-18).

3.1.1 Adaptation of a Virtual Platform at Resource Demand Fluctuations

To show how an application controller can adapt to changes in application resource allocation using the vertical scaling policy due to price variation, we ran a micro-benchmark using an MPI application, Zephyr [53]. Zephyr is a fluid dynamics simulation which runs for a user-defined number of iterations. Each iteration performs some computation, and if the application is started with multiple processes, also data is exchanged among them. Zephyr receives as input a configuration file and it simulates a volume filled by fluid for a specified simulated time, Zephyr periodically writes in a log file the following information: the CPU time for each iteration and the current number of iterations. For an SLO-driven Zephyr application, the application controller reads periodically the CPU time for each iteration and it compares it with a reference CPU time computed such that the application finishes its iterations at the user's given deadline.

We started an SLO-driven application and four best-effort applications (Zephyr applications running with a controller that does not change the bid during the application runtime) on a node, each in a VM with 4 cores. The SLO-driven application has a budget of 60000 credits, from which it can spend as much as it wants, while the other best-effort applications start with a bid of 100 CPU credits and 900 memory credits which remains unchanged during their execution. For this setup we used a node from our cloud. The SLO-driven application controller was started at the beginning of the experiment, while the other four application controllers were started during its execution. The last best-effort application was submitted after 20 minutes from the experiment start. The SLO-driven application has an ideal execution time of 77.5 minutes.

For clarity, we first run the Zephyr application with a best-effort controller instead of a SLO-driven one. Figure 5 shows the progress the application makes over time, i.e., its iteration execution time: (i) when it runs alone on the node (the ideal iteration time); (ii) and after the other applications are submitted (the best-effort iteration time). The performance difference in this case is highly noticeable: after all applications started executing, the iteration execution time increased almost 10 times. This degradation is not only due to its reduced resource allocation but also due to the other applications.

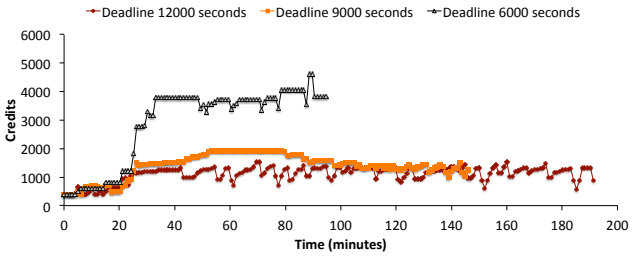


Figure 6: CPU bid variation for three different deadlines.

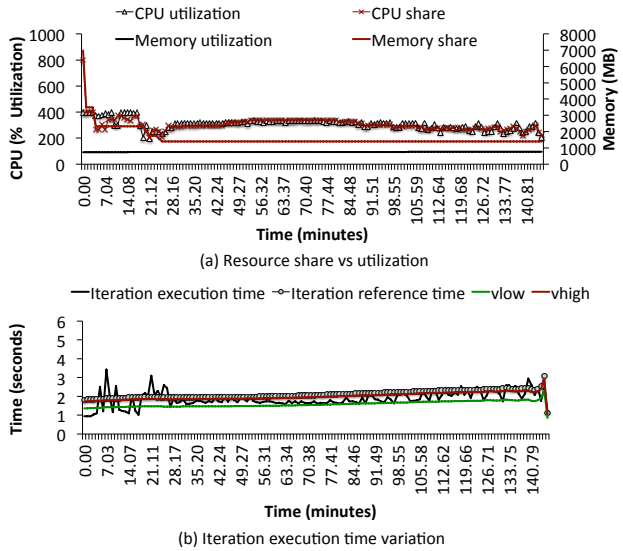


Figure 7: Application resource and performance variation when running with a deadline of 9000 seconds: (a) shows the variation in resource allocation and utilization and (b) shows how the controller keeps the perfomance, defined as iteration execution time, between the defined limits.

Then we ran the application with three different deadlines: 12000 seconds, 9000 seconds and 6000 seconds. We repeated each experiment three times and computed the average of the obtained values.

Figure 6 shows the bid of the SLO-driven controller for CPU resource for the different application deadlines. The bid stabilizes after all the applications were submitted. The application with the smallest deadline demands a maximum allocation and thus, the submitted bid is also much higher than in the other two cases.

Figure 7(a) shows the variation in the resource share, and respectively utilization for the application when it is run with a deadline of 9000 seconds. As the behavior is similar for the other two tested deadlines, we ommit depicting them. Remember that the VM resource share is the proportional-share the VM gets according to its bid. The resource utilization is how much the application inside the VM consumes. The left axis shows the CPU resource, i.e., percent-

age of total CPU time, while the right axis shows the memory resource, i.e., in MBs. The best-effort application arrivals can be noticed by looking at the changes in the SLO-driven VM resource share. In this case, the memory share reflects the best these arrivals. When the application started alone on the node the VM's share is the entire node capacity. After each application arrival, this share decreases until it equalizes the VM utilization. After all the best-effort applications started running, the SLO-driven controller keeps a reduced CPU share as the application can meet its deadline.

Figure 7(b) shows the variation in the SLO-driven application iteration execution time. To understand the behavior of the vertical scaling algorithm, we also give the lower and upper scaling thresholds, v_{low} and v_{high}. We notice that after all the best-effort applications started running, the application controller manages fairly well to keep the iteration execution time between these two thresholds. However, there are cases in which the iteration execution time oscillates. We think that one cause for this variation is the sharing of physical cores between more VM processes. The SLO-driven application receives more CPU than the best-effort applications, and thus less of the VMs in which these best-effort applications run get scheduled on the same physical cores as its own.

3.1.2 Adaptation of a Virtual Platform at SLO Modification

Merkat's application controller can also react to a user imposed condition. To show how it does so, we ran a SLO-driven Zephyr application and changed the user-specified deadline during the application execution. We submitted a deadline-driven application and four best-effort applications to Merkat. All the applications are started on the same node, each in a VM with 4 cores. The best-effort applications are started in the first few minutes after the start of the deadline-driven application. The SLO-driven application has a budget of 60000 credits, from which it can spend as much as it wants, while the other best-effort applications use a fixed bid of 100 CPU credits and 900 memory credits per VM.

The SLO-driven application is started with an initial deadline of 12000 seconds. After 64 minutes from the application start the deadline is changed to 9000 seconds. The application controller's bid adaptation and the allocation changes can be noticed in Figure 8. Figure 9 shows the application controller behavior and the variations in the estimated application execution time due to allocation and bid fluctuations. The additional submitted best-effort applications at the begining of the experiment leads to a decrease in application allocation and thus an increase in its execution time. However, the application controller doesn't react aggressively as its allocation, in this case 266 CPU units, is enough to meet the application deadline. When we decrease the application's deadline (noticed in Figure 9 from the change in the reference time), the application controller also adjusts the bid aggressively, leading to an increased resource allocation, in this case close to 400 CPU units, which is the maximum VM allocation. This increase allows the application to keep its iteration execution time close to the reference, thus meeting its deadline.

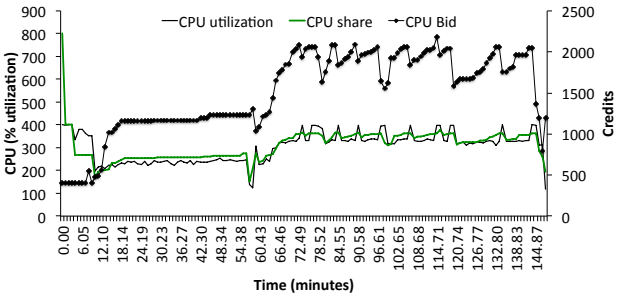


Figure 8: Application's CPU utilization and bid variation due to adaptation to the new deadline.

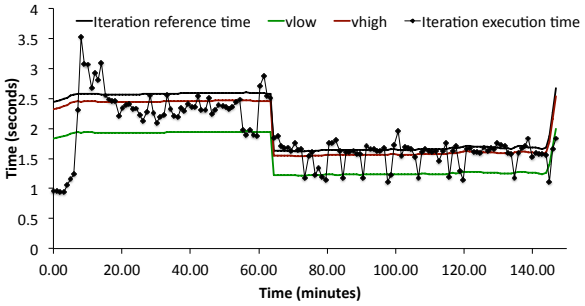


Figure 9: Application's iteration execution time variation due to adaptation to the new deadline. The reference iteration execution time and the two thresholds used in Algorithm 2 are also shown.

3.2 A Horizontal Scaling Policy

The horizontal scaling policy controls the number of VMs and their resource payments. The horizontal scaling policy outputs the number of VMs and the bids for the next time period based on the following information: (i) current resource prices, number of VMs and bids; (ii) the VM maximum allocation; (iii) application performance metric, upper and lower performance threholds. Algorithm 3 details this policy.

The algorithm is composed of two steps: (i) first, a new number of VMs, $nvms$, is computed based on the application performance metric; (ii) then, this number is compared to and limited to a maximum number of VMs computed for the next time period so that all the VMs receive an amount of CPU and memory equal to their maximum resource utilization, a_{max}, at the current resource prices.

The first step, detailed in method $HorizontalAdaptation$ (Lines 2-6), is used *to preserve the budget when the SLO is met* or *to increase the resource demand when the SLO is not met*. The second step, detailed in method $GetUpperBound$ (Lines 8-12), is used to reduce the resource demand *when the current budget is not enough to meet the SLO*. For this last step, the bid value for each VM resource is easily computed from the Equation 1 by replacing a_i with $N \cdot a_{max}$, where N is the number of VMs and a_{max} is the maximum resource utilization

Algorithm 3 Horizontal scaling policy

HorizontalAdaptation $(P, nvms_{old}, bid_{max}, alloc_{max}, v, v_{low}, v_{high})$
if $v > v_{high}$ **then**
  $nvms \leftarrow nvms + 1$
if $(v < v_{low}$ **then**
  pick vm to release
  $nvms \leftarrow nvms - 1$
$(N, bid) \leftarrow$ **GetUpperBound(**P**,** $nvms$**,** bid_{max}**,** $alloc_{max}$**)**
if $N < nvms$ **then**
  release (nvms - N) VMs
else
  request new $nvms - nvms_{old}$ VMs
return $nvms$

GetUpperBound(P**,** $nvms_{max}$**,** bid_{max}**,** $alloc_{max}$**)**
$resources \leftarrow \{cpu, memory\}$
find maximum value of N $\in [1, nvms_{max}]$ for which $\sum_{r \in resources} \cdot \frac{P[r] \cdot alloc_{max}[r] \cdot N}{1 - \frac{alloc_{max}[r] \cdot N}{Capacity[r]}} < bid_{max}$
$nvms \leftarrow N$
$bid[r] \leftarrow \frac{P[r] \cdot alloc_{max}[r] \cdot N}{1 - \frac{alloc_{max}[r] \cdot N}{Capacity[r]}}$, $r \in resources$
return (nvms, bid)

per VM. To find the upper bound on the number of VMs, the algorithm performs a binary search between 1 and $nvms$ by checking at each iteration if the sum of bids the controller needs to submit is less than its budget.

3.2.1 Adaptation of Two different Virtual Platforms

Let us illustrate the use of such a policy in a scenario in which a cluster needs to be shared between two commonly used frameworks: Condor [25] and Torque [40]. Users submit applications to each framework. These frameworks are usually used by scientific organizations to manage their HPC clusters. Each framework has its own scheduler which puts applications in a queue and runs them on nodes when resources become available for them. We implemented a scaling policy for each framework that uses Algorithm 3 to provision VMs. For Torque applications, the policy minimizes the wait time in queue, while for the Condor applications the policy maximizes the throughput. In both scaling policies, the framework copes with fluctuations in price in two ways. First, it avoids requesting a large number of VMs per time period, as provisioning them is wasteful if the price increases in the next period. Second it avoids starting new VMs if some VMs were released due to a price increase in the previous period. We deployed the Condor framework to process parameter sweep applications and the Torque framework to process MPI applications; both application types are commonly used at EDF. Then, we have studied how Merkat adapts the resource demand of each framework based on its workload.

We submitted 61 Zephyr applications to Torque with execution parameters taken from a trace generated using a Lublin model [26]: the number of processors was generated between 1 and 8 and the execution time had an average of 2479 seconds with a standard deviation of 1243.5. We submitted 8 parame-

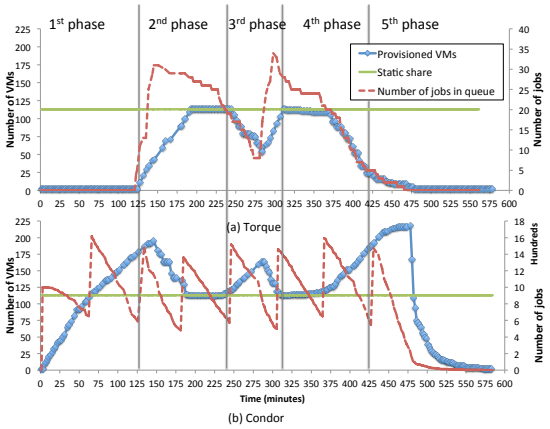


Figure 10: The variation of provisioned VMs versus queued jobs for each framework.

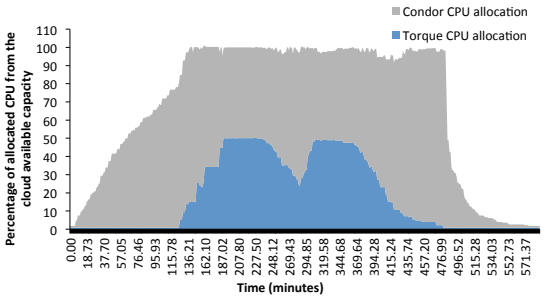


Figure 11: The variation of CPU allocation for both frameworks, as a percentage of the total available CPU capacity of the cloud. We omitted the variation of memory allocation as it is similar.

ter sweep applications composed of 1000 jobs of one processor each to Condor with an inter-arrival time of one hour. As we did not have access to a real parameter sweep application, we used the stress benchmark, which ran a CPU intensive worker for different execution time intervals generated with a Gaussian distribution. The average task execution time was 478 seconds, with a standard deviation of 363. Both frameworks were deployed on 32 nodes managed by Merkat and receive an equal budget.

Figure 10 shows the number of running and queued jobs in the Torque/Condor's queues and the number of VMs provisioned over time. If each framework is assigned an equal share of the infrastructure, it would obtain a maximum of 112 VMs (green line from Figure 10).

We divided our experiment in five phases. In the first, third and fifth phase Merkat's controller is capable to take advantage of the under-utilization periods of the infrastructure and provision up to as many VMs as allowed by the infrastructure's capacity for the Condor framework. In the second and fourth phase, both frameworks need more resources than their fair share to process their workload, thus each of them provisions an equal number of VMs. For

clarity, Figure 11 shows the total CPU allocation for each framework, as a percentage from the total available cloud capacity. We have omitted the memory allocation, as it exhibits a similar trend. In contention periods, each framework receives half of the available CPU while in under-utilization periods, the Condor framework scales its CPU allocation according to its workload demand.

This experiment shows that the horizontal scaling policy allows applications to expand and shrink their resource demand according to the resource availability of the infrastructure, thus leading improving the resource utilization.

3.3 Hybrid Scaling

When needed, the previous policies can be combined. For example, the application running under a deadline constraint, can first scale its resource demand vertically. Then, if the deadline is not met and the resource demand cannot be scaled vertically anymore, e.g., it reached the maximum resource utilization a_{max} of one VM, the algorithm switches to the horizontal scaling policy.

3.4 Discussion

We have seen that the virtual platforms react well to both changes in system workload and user requirements. Given that the system is not highly dynamic, and the application controller has time to adapt, the application can run with a smaller resource allocation and optimize its budget. The application controller optimizes the application execution cost while allowing other applications with less budget to use resources. However, our previously-presented policies do not cover all the possible cases. For example, the user's deadline might be too strict for the current price. In this case, the application controller might need to estimate the future infrastructure load and send feedback to the user regarding the minimum deadline it can meet. Starting from the simple mechanisms presented here, to improve the SLO support more complex policies can be developed, based on application profiling and price prediction.

Finally, in both our policies, the controllers use VMs with a predefined maximum size. This size can be set up by the user, if for example she knows that her application will never use more than that, or it can be configured by the cloud infrastructure, e.g., as the capacity of the node. Nevertheless, the VM resource allocation changes dynamically, as the vertical scaling policy can steer it during the application runtime.

4 Implementation and Performance Evaluation

In this Section we analyze the performance of Merkat in terms of total user satisfaction when applications adapt their resource demands to track a given SLO and show its flexibility in supporting different user types. User satisfaction is an important metric regarding the performance of a resource management system. This satisfaction depends wether that the user's SLO is violated or not and on how much the user actually valued the execution of her application.

We have implemented a prototype of Merkat and validated it through simulation and on the Grid'5000 testbed [7]. We used simulation to test Merkat's

algorithms with a large workload. Running a large workload gives a better insight in the total satisfaction that the system can provide. Due to time and resource limitations, running a large workload on Merkat in a real environment, which lasts for days, or possibly weeks, and with a large number of nodes, would have been unpractical. We discuss next the implementation of Merkat. Then we describe the results we obtained from measuring the total user satisfaction provided by Merkat in simulation and on a real-world testbed.

4.1 Performance Results from Simulation with Large Traces

We measured the performance of our system in terms of total user satisfaction in different contention scenarios, compared to two traditional resource allocation policies: First-Come-First-Served (FCFS) and Earliest Deadline First (EDF). The first policy is usually applied by IaaS cloud managers to schedule VMs. It keeps the requests in a queue and schedules them when resources become available. The second policy is used to minimize the number of missed deadlines. The requests are ordered in the queue based on their remaining time to deadline and requests with the smallest remaining time to deadline are executed first. Other deadline-based algorithms are available in the state of the art, but a large majority is specific to one application type, e.g., bag of tasks, workflows, environment, e.g., public clouds, or are offline solutions. Moreover, it is difficult to choose among these different algorithms the most representative one. Thus, we chose the EDF policy as it is well-known in the state of the art and is often used as a comparison baseline. It is important to note that cloud managers cannot practically apply EDF or similar algorithms without limiting their support to a predefined set of application goals (e.g., meeting deadlines). Nevertheless, we wanted to compare our system with a centralized system that targets a fixed type SLO.

4.1.1 Simulation Setup

We implemented the algorithms of Merkat in CloudSim [8], an event-driven simulator implemented in Java. In our case, we simulate the datacenter, the VM Scheduler and multiple applications, created dynamically during the simulation. Applications are created according to their submission times, taken from a workload trace, and are destroyed when they finish their execution. In our simulator there is no distinction between an application and its virtual platform. The application applies the resource demand adaptation policy by itself and interacts with the datacenter to change the bids for its VMs. As CloudSim does not model the cost of VM operations, we have also implemented a model for several VM-related performance overheads: memory over-commit, VM boot/resume and VM migration [14].

Application Model We consider batch applications composed of a fixed number of tasks running in parallel. To finish their execution, applications need to perform a certain computation amount (e.g. 1000 iterations). These applications have a relatively stable iteration execution time. The iteration execution time depends on the resource allocation received by each task. For example, if each task receives one full core, one iteration takes 1 second. If the resource allocation drops at half, the same iteration takes 2 seconds. We

simulate these applications as sets of tasks, with each task requiring one CPU core and a specified amount of memory.

User Model To measure the total user satisfaction, we model the user satisfaction as a utility function of the budget assigned by the user to its application and the application execution time. There are different functions which can be used to model this satisfaction, and they depend on the behavior of the user. In this paper we define several functions, derived from discussions with scientists at Electricité de France:

- **Full deadline users** A common case is of an user who wants the application results by a specific deadline. If the application doesn't finish its execution at the deadline, the user is unsatisfied.
- **Partial deadline users** Some users might value partial application results at their given deadline; for example, for a user who implemented a scientific method and needs to run 1000 iterations of her simulation to test it, finishing 900 iterations is also sufficient to show the good method behavior.
- **Full performance** Finally, other users want the results as soon as possible, but they are also ready to accept a bounded delay. The upper bound of the delay is defined by the application deadline. For example, a developer wants to test a newly developed algorithm. She wants the results as fast as possible, but if the system is not capable to provide them, she might be willing to wait until the morning.

Before discussing the signification of utility functions, we define the following terms. t_{exec} is the application execution time. $t_{deadline}$ is the time from the submission to deadline. t_{ideal} is the ideal execution time, i.e., if the application runs on a dedicated infrastructure. $work\_done$ represents the number of iterations the application managed to execute until it was stopped and $work\_total$ represents the total number of iterations. B is the application's budget per budget renewal period and per task. B is assigned by the user and reflects the application's importance.

Table 1 summarizes the used utility functions. The full deadline user values the application execution at her full budget if the application finishes before deadline. Otherwise, we express her dissatisfaction as a "penalty", which represents the negative value of her budget. The partial deadline user is satisfied with the amount of work done until the deadline. Thus the value of her satisfaction is proportional to this amount. The full performance user becomes dissatisfied proportionally to her application execution slowdown. We bound the value of her dissatisfaction at the negative value of her budget.

User Type	**Utility Function**
full-deadline	B, if $t_{exec} \leq t_{deadline}$, -B otherwise
partial-deadline	B, if $t_{exec} \leq t_{deadline}$, $B \cdot \frac{work\_done}{work\_total}$ otherwise
full-performance	B, if $t_{exec} = t_{ideal}, max(-B, B \cdot \frac{t_{deadline}+t_{ideal}-2 \cdot t_{exec}}{t_{deadline}-t_{ideal}})$ otherwise

Table 1: Utility functions.

Application Policy For each user type we derive an application-specific execution policy from the vertical scaling policy, presented in Section 2.3, as follows:

- **Full deadline:** The policy is derived from the Algorithm 2. Applications start when the price is low enough to ensure a good allocation. During their execution they adapt their bids and use suspend/resume mechanisms to keep a low price in low utilization periods and to use as much resource as their SLO allows in high utilization periods. If, during its execution, the application sees it cannot meet the deadline it stops.
- **Partial deadline:** This policy is similar to the previous policy but, nevertheless, there are two differences: (i) the application suspends when a minimum allocation cannot be ensured (e.g., 30% cpu time or 30% physical allocated memory); (ii) as any work done at the deadline is useful, the application always runs until its deadline
- **Full performance:** This policy is designed for full-performance users. The policy is similar to the **Full deadline** policy. Nevertheless, during its execution, the application, instead of tracking a performance reference metric, it tracks a reference allocation defined as the maximum used by the application VM. When the application cannot have a minimum allocation at the current price, the application suspends.

4.1.2 Workload

To evaluate the system performance we use a real workload trace. Real workload traces are preferred to synthetic workloads as they reflect the user behavior in a real system. Such traces are archived and made publicly available [5]. As a workload trace, we chose the HPC2N file. This file contains information regarding applications submitted to a Linux cluster from Sweden. The cluster has 120 nodes with two 2 AMD Athlon MP2000+ processors each. We assigned to each node 2 GB of memory. The reason for choosing this trace is the detailed information regarding memory requirements of the applications. Nevertheless, this information was not specified for all the applications. Thus, for applications with missing memory requirements, we assigned a random amount of memory, between 10% and 50% of the node's memory capacity. We ran each experiment by considering the first 1000 jobs, which were submitted over a time period of 18 days. We scale the inter-arrival time with a factor between 0.1 and 1 and we obtain 10 traces with different contention levels. A factor of 0.1 gives a highly contended system while a factor of 1 gives a lightly loaded system.

We consider that all applications have a deadline and a re-chargeable budget. As we couldn't find any information regarding application deadlines, we assigned synthetic deadlines to applications, between 1.5 and 10 times the application execution time. We assume that the budget amount the user wants to pay depends on the application's deadline: a user with a less urgent application wants to pay less. Thus, the budgets assigned to applications are inversely proportional to the application's deadline factor, and computed from a base budget of 2000 credits per time period.

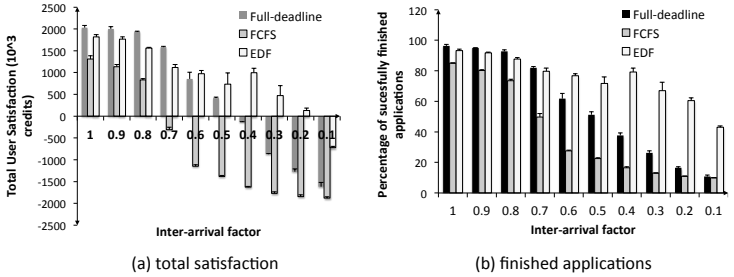


Figure 12: Proportional share market performance in different contention scenarios in terms of: total user satisfaction (a) and percentage of successfully finished applications (b). The contention increases from left to right.

4.1.3 Results

To see how the performance is influenced by the contention level, we measure the obtained satisfaction for each of the 10 obtained workload traces and for full deadline users. We repeated each experiment four times and the results are presented as the mean plus standard deviation.

Figure 12 (a) describes the results of our comparison. For clarity, Figure 12 (b) describes the number of applications that succesfully finished their execution until their deadline. We notice three aspects: (i) our system outperforms FCFS in all cases, as FCFS does not consider application valuation or SLO in its decisions. (ii) when the contention is not high, despite reaching almost the same number of finished applications as EDF, our system still outperforms it in terms of user satisfaction; (iii) however, when the system is highly loaded, its performance degradation increases. The performance gap between our mechanism and EDF can be explained as follows. When the contention is low, our system provides higher satisfaction than EDF, due to its fine-grained allocation policies. When the contention is high, more applications arrive at the same time and, EDF is capable to take better scheduling decisions: thus more applications with smaller deadlines, get to run on time. Because the deadline urgency is reflected in the application's value, EDF also leads to higher user satisfaction. In the case of our system, applications do not take the best allocation decision, as they adapt independently with only limited information. This decentralization leads to a loss in performance, compared to EDF. The performance degradation is the "price" payed by the nature of our system, which allows applications to behave selfishly.

We also measured the total user satisfaction when users have different models for the satisfaction they get from their application execution. Figure 13 describes the proportional-share market performance for each of the different previously discussed user models, in terms of total user satisfaction and, for clarity, number of successfully finished applications (i.e., their deadlines were met). Figure 13(b) describes the total satisfaction that our system provides to users when applications use the each of the three different policies. The best satisfaction is provided by the partial deadline policy, as users still gain a positive value from getting the results at their deadline, despite of not having all of them. The worse satisfaction is provided by the full performance policy, as users

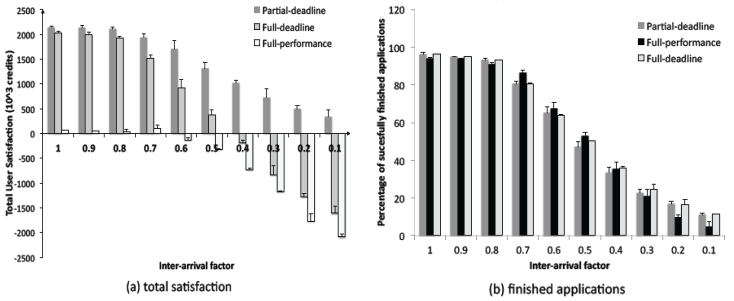


Figure 13: Proportional share market performance for different user models in terms of: total user satisfaction (a) and percentage of successfully finished applications (b). The contention increases from left to right.

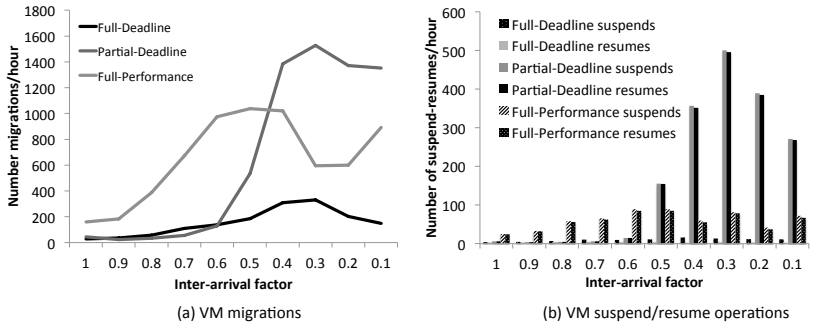


Figure 14: Number of VM operations for each user utility policy and for different contention levels. The contention increases from left to right.

are also more demanding: they get dissatisfied really fast and they perceive a negative value if their jobs don't finish at their deadline. Figure 13(a) shows the percentage of finished applications for each policy. It is interesting to see that in all cases, the percentage remains almost the same. Nevertheless, fewer applications finish in the case of partial deadline users, as applications are not stopped before their deadline.

As the application adaptation algorithms lead to VM operations, we measure their cost. We keep the same settings from the previous experiments and we record the number of VM operations during the experiment run. Figure 14 describes the average number of VM operations per hour performed by the VM Scheduler for different user utility functions. We notice that the Full-Deadline policy is the most effective as it has the least number of VM operations. This is due to the fact that with this policy the applications are less aggressive in acquiring resources. We notice that in the most highly loaded periods the number of migrations decreases. This is explained by the fact that when there is a high load, many applications don't resume and even don't start their execution as their deadline cannot be met. The Partial-Deadline policy has the most migrations, number which increases with the contention, as even in highly contended periods applications continue running with smaller resource amounts. Moreover, applications start executing whenever they can get a small resource amount.

This leads to more bid adaptations, and thus resource allocation changes and more migrations. The number of suspend/resume operations follows the same trend. The reason behind this is that applications resume at a small allocation, as the policy considers that any progress the application makes is useful. The Full-Performance policy follows a similar trend as the Partial-Deadline policy but the number of migrations becomes higher even when there is not much contention. This is explained by the aggresivity of the application adaptation policy that always increases the bid to get as much resource as possible.

The results of these experiments show that Merkat can accomodate different user types on the same infrastructure while providing good satisfaction.

4.2 Performance Results from Testbed Experiments

To measure the user satisfaction that Merkat provides on a real testbed, we set up a cloud managed first by Merkat and then by a batch scheduler, e.g., Maui [27]. Maui is often used by HPC organizations to manage their clusters. To run applications, Maui uses algorithms based on FCFS, with backfilling.

4.2.1 Merkat Prototype and Testbed

Merkat is implemented in Python and depends on the Twisted [42], paramiko [31] and ZeroMq [54] libraries. We used the Twisted framework to develop the XML-RPC services. Paramiko is used for the VM connection: the application management system needs to test and configure the VMs in which the application runs and it does so through SSH. ZeroMq is used for the internal communication between various components of Merkat (e.g., the Applications Manager and application controllers, or the application controllers and virtual cluster monitors). Merkat's services store their state in a database storage for which we have used a MySQL server. The connections to the VMs are done through SSH in parallel, by using a thread pool with a size defined in the configuration file. As an IaaS Cloud Manager we use OpenNebula [39] 4.6. Merkat's services communicate with OpenNebula through its XML-RPC API. We also shut down OpenNebula's default scheduler, and replaced it with our VM Scheduler.

We deployed both Merkat and Maui on a cloud of 10 compute nodes on the edel cluster of the Grenoble site from Grid'5000. When using Maui, we deploy the VMs and add them to Maui as physical nodes at the begining of the experiment. Each node has 2 Intel Xeon E5420 QC processors (4 cores) and 24GB of memory. The VM Scheduler of Merkat assigns for each node an available capacity of 700 CPU units and 23 GB RAM (one core is reserved for the hypervisor/node's operating system) and it distributes it among the VMs running on the node. All nodes are connected through a Gigabit Ethernet and an Infiniband link. We use the Gigabit Ethernet link for VM communication and the Infiniband link for VM migration. The VM images are stored on a NFS server. To speed-up the VM deployment we use copy-on-write VM images. Merkat's VM Scheduler uses a scheduling period of 60 seconds and the application controllers read the application's performance metrics at every 80 seconds. We wanted a scheduling period that is small enough to allow applications to adapt fast their bids, and large enough to allow the VM migrations to finish. Unless otherwise specified, all VMs have a maximum capacity of 100 CPU units

and 900MB of RAM. Each VM is configured with Debian Squeeze 6.0.1 OS and KVM [22] as a hypervisor on each node.

4.2.2 Workload

We run 160 Zephyr applications, with 1 to 8 processes, each process in one VM and we consider full deadline users. As using a real workload trace would have lead to longer experiment execution times, which would not have been possible on Grid'5000, we used a Lublin [26] model to generate the workload. We chose this model as it is realistic. The generated application budgets were within the range of 1000 and 28700 credits per scheduling interval.

4.2.3 Results

We repeated the experiment 4 times and we averaged the number of succesfully finished applications and the total satisfaction provided by Merkat and Maui. When Merkat is used to manage the cloud, 24.4% more applications finish until their deadline than when using Maui (in Merkat 82.5% of the applications finish until their deadlines compared to 58.1% when using Maui). Merkat also leads to more users being satisfied from running their applications on the cloud than traditional systems like Maui (in Merkat the total user satisfaction is 764330 credits, compared to a negative satisfaction of -178581 credits from Maui). This result provides evidence of the effectivness of Merkat in managing a real cloud.

5 Limitations and Future Work

In this section we discuss the remaining open issues of our approach.

Currency Policies Designing currency policies to decide how to give budgets to users remains a difficult task. When designing Merkat we encountered two issues: (i) deciding the total credit amount that circulates in the system; (ii) deciding the time period over which the user budgets are renewed.

The value of the total credit amount influences the performance of the system. If the total credit amount is too large, the resource prices become too inflated and lead to poor user satisfaction. If the credit amount is small and new users arrive, current users might be funded at a rate too small to allow them to run highly urgent applications. In Merkat setting this amount can be done by the infrastructure administrator based on the number of users in the system, the price history and the capacity of the infrastructure. If the administrator observes that the price increased considerably in the last period, she can reduce the total credit amount. If new users arrive, she can increase this amount. Designing an automatic system to manage this amount remains an open issue. Such a system needs to adapt the amount based on past price fluctuations, infrastructure size and feedback from the users regarding their resource usage versus assigned budget.

The time period over which the user budgets are renewed influences the user behavior. If the time period is small, users will not have incentives to save budget, and thus, to judiciously use resources. If the time period is too large, users might starve while resources are left idle. We consider that such a period

should be in terms of weeks or several months. However, deciding properly this time period remains an open issue as it requires running the system on a real-world infrastructure and getting feedback from users.

System Stability Poor adaptation policies, currency management policies or a too small scheduling interval might lead to system trashing, i.e., the system would spend too much time adapting, wasting resources and leading to poor application performance. The system's stability can be improved in two ways: (i) increasing the VM scheduler's scheduling period, together with the application's controller's adaptation period; (ii) improving the adaptation policies. The first method will lead to a less reactive system, while the second method requires sophisticated algorithms for price and application performance prediction. Designing such prediction algorithms remains an open research question.

Scalability As the current scale on which Merkat was tested is quite limited, also due to limitations in the underlying third party software stack, we could not identify the upper scalability limit of Merkat. We believe that this upper limit is determined by two factors: (i) the performance of the VM Scheduler in allocating resources when it has to cope with adaptation requests from a large number of applications; (ii) and the performance of the IaaS manager. While the performance of the IaaS manager is outside the scope of this work, the performance of the VM Scheduler can be further improved by optimizing the VM allocation and placement algorithm.

Topology-awarness Co-location and migration might lead to performance degradation for running applications. Especially, in the case of HPC applications, care must be taken in placing communicating application processes as close as possible to each other. Currently, Merkat does not consider the application's or the cluster's topology. However, it can be further extended to allow users to express and pay for running their application processes as close as possible to each other.

I/O resources The current resource allocation algorithms could be extended to consider network and storage resources. Such resources can become a bottleneck when network or data intensive applications are executed. Thus, it is normal to make them available at a price that reflects the total resource demand. Regarding the network resource, software tools can be used to limit the bandwith available to each virtual machine and ensure a proportional share. We envision that similar mechanisms could be applied for storage too.

High Availability To provide users with a production system, Merkat services require self-healing capabilities to make the platform highly available. State-of-the art solutions [21] can be used to achieve these goals.

6 Related Work

We classify the related work in three categories: (i) resource management systems for clusters; (ii) platform as a service systems for clouds; (iii) and resource management systems that apply a market.

6.1 Resource Management Systems for Clusters

Several resource management systems were developed to support the execution of dynamic applications on shared infrastructures. To support different application types on the infrastructure, earlier systems focus on sharing the cluster or grid resources [9, 34, 23] between frameworks (e.g., Torque, SGE) using virtualization. However, in most cases, the policies to decide the resource allocations for each framework are missing. Mesos [20] is a system that allocates resources to frameworks based on the concept of "resource offer", i.e., a list of available resources on nodes. Mesos gives resource offers to frameworks while frameworks can filter the offers and decide what resources to accept. Omega [36] presents frameworks with a "replicated" shared view of the cluster, called a "cell state". Each framework keeps its own cell state, which is synchronized periodically to reflect allocation changes, and selects from it what resources to use. In Yarn [43] each application has its own application manager that requests resources from a global scheduler, which allocates resources by considering specific application constraints. In contrast to Merkat, all these systems don't consider individual application performance or SLOs.

In the datacenter community there are different systems to allocate resources between different, usually web-based, applications [49, 56, 46, 19, 28]. They rely on the use of performance models to obtain estimations and predict the application resource allocation to minimize the number of used cluster nodes. In this case, resources are allocated to VMs in a fine-grained fashion as applications like web servers can have varying CPU and memory utilization. VM migrations are employed to move the VMs among nodes either reactively [48], or proactively [37]. The closest to our work is "Friendly virtual machines" [55]. In this work the resource control is fully decentralized: applications adapt autonomously their resource demand using a "feedback signal" received from the system. However, the authors assume that applications are altruistic. In Merkat, we consider that applications are selfish and application controllers adapt their application resource demand based on the current resource price and application budget.

Quasar [15] is a resource manager, which uses classification techniques to determine the amount of resources required by each application to meet a specified performance. In Quasar there are only two types of workoalds: best-effort and SLO-driven. In contrast to Quasar, Merkat uses a market to distribute resources, leading to a more fine-grain differentiation between application priorities.

6.2 PaaS Systems

Many PaaS systems, both commercial [3, 6, 12] and research [32] provide runtime support for applications hiding from users the complexities of managing resources. These systems, however, provide typically closed environments, forcing users to run only specific application types (e.g., web, MapReduce). If new programming frameworks appear, the PaaS provider needs to first develop the necessary support on the infrastructure, and then offer to the users the possibility to use it. In contrast to these systems, Merkat allows users to run new application types while distributing the resources among them based on their value.

Similar to Merkat, Meryn [16] is a PaaS that supports new application types through a decentralized resource control: resources are allocated among frameworks with consideration to the SLO and cost of the applications managed by them. Merkat is different from Meryn in two ways: (i) it allocates virtual environments per application while Meryn allocates them per framework; (ii) and it focuses on managing contention on a private infrastructure by implementing a virtual economy while Meryn uses cloud bursting to offload its workload in public clouds while optimizing the PaaS provider's profit.

6.3 Market-based Resource Management Systems

Using a market to manage the resources of a distributed infrastructure is a well-studied problem in the context of clusters and grids [50]. The reason why markets became so popular in the distributed systems community is the notion of *cost*, which makes users more aware of how many resources to acquire for their own use. By using a market, users have incentives to assign the right priorities to their applications. Thus, applications with urgent resource needs can get resources in time even in high load periods, while applications with less urgent resource needs run in low load periods. Based on the pricing model, there are two commonly used market models: commodity markets and auctions.

In commodity markets the resource price is established using demand and supply functions and both consumers and providers buy and respectively sell at this price. These market types rely on tatonnement algorithms to adjust the resource price to reduce the excess demand close to zero [10]. These algorithms either use estimations of the excess demand, as in G-commerce [47], or they rely on the participants to send their demand as a function of price like in On-Call [29]. For scalability reasons, resources are allocated in a coarse-grain way, i.e., in terms of CPU slots or number of nodes. One particular case is Libra [38], which allocates CPU proportionally to the application's deadline on each node while a global pricing mechanism, based on the infrastructure utilization, is used to balance supply with demand. The use of the proportional-share policy maximizes the infrastructure utilization, as opposed to a case in which applications are allocated exclusive-access to nodes. Admission control ensures that the applications accepted in the system do not lead to deadline misses for the other applications. However, the same admission control mechanism might prevent urgent applications for running while less urgent applications occupy all the resources. Aneka [52] implements a dynamic pricing model too by using advance reservations as a substrate resource allocation model. Unfortunately, as the price is set at the beginning of the execution, applications coming in the system in a low demand period will be charged with a small price, while urgent applications coming later, might not get all the resources they need for their execution.

Auctions establish the resource price based on how much users are willing to pay for resources. Auctions clear the market faster than the price computation algorithms from the commodity markets, allowing users with the most urgent demand to get their resources with minimum delays. Multiple attempts were made to use auctions to schedule static MPI applications, on clusters [41, 51] or to run bag-of-tasks applications on grids [1]. In this case, users bid for an application execution and the scheduler decides which application gets to run through the auction. An auction, which assigns resources to the highest bidders, ensures

that the most valuable applications are running on the infrastructure. Nevertheless, these proposed systems address the case of applications with known execution times and static resource requirements.

A first step in providing more flexibility for dynamic applications was taken by Popcorn [33] and Spawn [44], in which applications composed of many tasks can shrink when the resource price is high and expand when the resource price is low. Systems like Tycoon [24] or REXEC [11] implement a proportional-share policy per node to allocate fractional amounts of CPU. Ginseng [2] is a cloud platform that uses an auction to allocate memory to VMs on a node. The applications running in VMs have to decide the quantity of memory and the bid based on their performance. Although it represents a first step towards a market-based platform, further development is required to provide users with a complete system. A dynamic priority scheduler is proposed for Hadoop [35], to allocate map/reduce slots to applications using proportional-share. Users are assigned a budget and they spend it to run applications on the cluster, by specifying a spending rate, i.e. the user's willingness to pay for a slot and for a time period. The budget and the spending rate allow the user to control how many slots get assigned to her applications over time, and thus, to control the application execution time. These systems do not provide support for different SLOs, as, more specifically, they do not monitor and adapt the application resource demand on the market. In contrast to these systems, Merkat controllers can adapt applications in two different ways, vertical and horizontal, to meet user SLOs.

7 Conclusion

This paper introduced a platform for application and resource management in private clouds, called Merkat. The goal of Merkat is to maximize the resource utilization of the managed infrastructure while providing support for different application resource demand models and user SLOs. To meet this goal Merkat transforms the organization's infrastructure to a private cloud and relies on: (i) a proportional-share market to allocate fine-grained CPU and memory amounts to VMs and to make users aware of the cost of using resources; (ii) a set of autonomous application controllers that can scale the application's resource demand vertically and horizontally to meet the user's SLO under current price and user budget constraints. Merkat supports different applications and user SLOs by decentralizing the resource control and treats contention periods that might appear on the private infrastructure by using market mechanisms.

We evaluated Merkat in simulation and on the Grid'5000 testbed. The obtained results show that: (i) Merkat is flexible enough to support different applications and SLOs; (ii) Merkat can adapt the application resource demand to the infrastructure load and application's SLO, maximizing resource utilization while leading to a better user satisfaction; (iii) the resource control decentralization has a reasonable performance impact compared to centralized resource management systems.

8 Acknowledgements

This work was done while the first author was a PhD student at INRIA and EDF R&D and was supported by ANRT through the CIFRE sponsorship No. 0332/2010. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

[1] D. Abramson, R. Buyya, and J. Giddy. A computational economy for grid computing and its implementation in the nimrod-g resource broker. *Future Generation Computer Systems*, 18(8):1061–1074, 2002.

[2] O. Agmon Ben-Yehuda, E. Posener, M. Ben-Yehuda, A. Schuster, and A. Mu'alem. Ginseng: Market-driven memory allocation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 41–52, New York, NY, USA, 2014. ACM.

[3] AmazonEBS. http://aws.amazon.com/.

[4] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Scale-up vs scale-out for hadoop: Time to rethink? In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 20:1–20:13, New York, NY, USA, 2013. ACM.

[5] P. W. Archive. http://www.cs.huji.ac.il/labs/parallel/workload/.

[6] M. Azure. http://www.windowsazure.com/.

[7] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and I. Touche. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 2006.

[8] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software Practice and Experience*, 41(1):23–50, 2011.

[9] J. S. Chase, D. E. Irwin, L. E. Grit, J. D. Moore, and S. E. Sprenkle. Dynamic virtual clusters in a grid site manager. In *Proceedings of 12th IEEE International Symposium on High Performance Distributed Computing, 2003.*, pages 90–100. IEEE, 2003.

[10] J. Q. Cheng and M. P. Wellman. The walras algorithm: A convergent distributed implementation of general equilibrium outcomes. *Computational Economics*, 12(1):1–24, 1998.

[11] B. N. Chun and D. E. Culler. Rexec: A decentralized, secure remote execution environment for clusters. In *Proceedings of the 4th International Workshop on Network-Based Parallel Computing: Communication, Architecture, and Applications*, CANPC '00, pages 1–14, London, UK, UK, 2000. Springer-Verlag.

[12] CloudFoundry. http://www.cloudfoundry.com/.

[13] S. V. Costache, N. Parlavantzas, C. Morin, and S. Kortas. Merkat: A market-based slo-driven cloud platform. In *Proceedings of IEEE International Conference on Cloud Computing Technology and Science*, CloudCom'13, 2013.

[14] S. V. Costache, N. Parlavantzas, C. Morin, and S. Kortas. On the use of a proportional-share market for application slo support in clouds. In *Proceedings of the 19th International European Conference on Parallel and Distributed Computing*, Europar'13, 2013.

[15] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, New York, NY, USA, 2014. ACM.

[16] D. Dib, N. Parlavantzas, and C. Morin. SLA-based Profit Optimization in Cloud Bursting PaaS. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Chicago, États-Unis, 2014.

[17] F. Glover, M. Laguna, et al. *Tabu search*, volume 22. Springer, 1997.

[18] A. Gulati, G. Shanmuganathan, A. Holler, and I. Ahmad. Cloud-scale resource management: challenges and techniques. In *Proceedings of the 3rd USENIX conference on Hot topics in cloud computing*, pages 3–3. USENIX Association, 2011.

[19] R. Han, L. Guo, M. M. Ghanem, and Y. Guo. Lightweight resource scaling for cloud applications. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID'12, pages 644–651. IEEE, 2012.

[20] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: a platform for fine-grained resource sharing in the data center. In *Proceedings of USENIX conference on Networked Systems Design and Implementation*, NSDI'11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association.

[21] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, pages 11–11, 2010.

[22] A. Kivity, Y. Kamay, and D. Laor. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, 2007.

[23] N. Kiyanclar, A. G. Koenig, and W. Yurcik. Maestro-vc: On-demand secure cluster computing using virtualization. In *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid Workshops*, 2006.

[24] K. Lai, L. Rasmusson, E. Adar, L. Zhang, and B. Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent and Grid Systems*, 1(3):169–182, 2005.

[25] M. Litzkow, M. Livny, and M. Mutka. Condor-a hunter of idle workstations. In *8th International Conference on Distributed Computing Systems*, 1988.

[26] U. Lublin and D. G. Feitelson. The workload on parallel supercomputers: modeling the characteristics of rigid jobs. *Journal of Parallel and Distributed Computing*, 63(11):1105–1122, 2003.

[27] Maui. http://www.nsc.liu.se/systems/retiredsystems/grendel/maui.html.

[28] M. Maurer, I. Brandic, and R. Sakellariou. Enacting slas in clouds using rules. In *Euro-Par 2011 Parallel Processing*, pages 455–466. Springer, 2011.

[29] J. Norris, K. Coleman, A. Fox, and G. Candea. Oncall: Defeating spikes with a free-market application cluster. In *Proceedings of the 2004 International Conference on Autonomic Computing*, ICAC'04, pages 198–205. IEEE, 2004.

[30] OpenStack. http://www.openstack.org/.

[31] paramiko. www.lag.net/paramiko/.

[32] G. Pierre and C. Stratan. Conpaas: a platform for hosting elastic cloud applications. *IEEE Internet Computing*, 2012.

[33] O. Regev and N. Nisan. The popcorn market. online markets for computational resources. *Decision Support Systems*, 28(1):177–189, 2000.

[34] P. Ruth, P. McGachey, and D. Xu. Viocluster: virtualization for dynamic computational domains. In *Proceedings of IEEE International Conference on Cluster Computing*, 2005.

[35] T. Sandholm and K. Lai. Dynamic proportional share scheduling in hadoop. In *15th Workshop on Job Scheduling Strategies for Parallel Processing*, 2010.

[36] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European conference on Computer systems*, Eurosys'13, 2013.

[37] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 5:1–5:14, New York, NY, USA, 2011. ACM.

[38] J. Sherwani, N. Ali, N. Lotia, Z. Hayat, and R. Buyya. Libra: a computational economy-based job scheduling system for clusters. *Software Practice and Experience*, 34:573–590, 2004.

[39] B. Sotomayor, R. Montero, I. Llorente, and I. Foster. An Open Source Solution for Virtual Infrastructure Management in Private and Hybrid Clouds. *IEEE Internet Computing*, 13(5):14–22, 2009.

[40] G. Staples. Torque resource manager. In *Proceedings of ACM/IEEE conference on Supercomputing*, 2006.

[41] I. Stoica, H. Abdel-Wahab, and A. Pothen. A microeconomic scheduler for parallel computers. In *Job Scheduling Strategies for Parallel Processing*, pages 200–218. Springer, 1995.

[42] Twisted. twistedmatrix.com/.

[43] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the2013 ACM Symposium on Cloud Computing*, SOCC'13, 2013.

[44] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, 1992.

[45] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 1. USENIX Association, 1994.

[46] Z. Wang, Y. Chen, D. Gmach, S. Singhal, B. Watson, W. Rivera, X. Zhu, and C. Hyser. Appraise: application-level performance management in virtualized server environments. *IEEE Transactions on Network and Service Management*, 6(4):240 –254, 2009.

[47] R. Wolski, J. S. Plank, T. Bryan, and J. Brevik. G-commerce: Market formulations controlling resource allocation on the computational grid. In *Proceedings of hte 15th International Parallel and Distributed Processing Symposium.*, pages 8–pp. IEEE, 2001.

[48] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Sandpiper: Black-box and gray-box resource management for virtual machines. *Computer Networks*, 53(17):2923–2938, 2009.

[49] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif. Autonomic resource management in virtualized data centers using fuzzy logic-based approaches. *Cluster Computing*, 11(3):213–227, 2008.

[50] C. S. Yeo and R. Buyya. A taxonomy of market-based resource management systems for utility-driven cluster computing. *Software: Practice and Experience*, 36(13):1381–1419, 2006.

[51] C. S. Yeo and R. Buyya. Pricing for utility-driven resource management and allocation in clusters. *International Journal of High Performance Computing Applications*, 21(4):405–418, 2007.

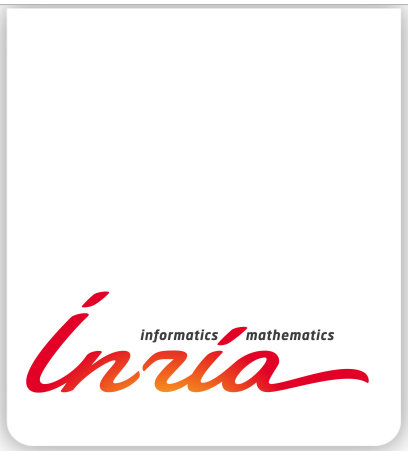
[52] C. S. Yeo, S. Venugopal, X. Chu, and R. Buyya. Autonomic metered pricing for a utility computing service. *Future Generation Computer Systems*, 26(8):1368–1380, 2010.

[53] Zephyr. https://github.com/kortas/zephyr.

[54] ZeroMq. http://www.zeromq.org/.

[55] Y. Zhang, A. Bestavros, M. Guirguis, I. Matta, and R. West. Friendly virtual machines: leveraging a feedback-control model for application adaptation. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 2–12, New York, NY, USA, 2005. ACM.

[56] X. Zhu, D. Young, B. J. Watson, Z. Wang, J. Rolia, S. Singhal, B. Mckee, C. Hyser, D. Gmach, R. Gardner, T. Christian, and L. Cherkasova. 1000 islands: an integrated approach to resource management for virtualized data centers. *Cluster Computing*, 12(1):45–57, 2009.



RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399